Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Sistemas DataSnap com SOAP

Crie sistemas integrados usando SOAP – Parte I

Nesse artigo veremos

- Implementação de uma estrutura SOAP e DataSnap para agilizar processos entre empresa e fornecedor;
- Apresentação de um cenário onde a mesma estrutura pode ser explorada;
- Levantamento de conceitos para entendimento da tecnologia;
- Estruturação do banco de dados;
- Demonstração de um ambiente baseado em camadas, utilizando o método de comunicação SOAP para trafegar os dados via protocolo HTTP.

Qual a finalidade

Desenvolver algo prático, que possa ser usado no dia-a-dia, aplicando os mesmos métodos em diversos sistemas.

Quais situações utilizam esse recurso?

Todo e qualquer sistema que necessita de visões on-line do cenário, podem ser construídos utilizando os conceitos que aprenderemos nesse artigo.

Maikel Marcelo Scheid

maikelscheid@gmail.com

É técnico em Informática com ênfase em Análise e Programação de Sistemas. Atua na área de Desenvolvimento de Softwares em Delphi para plataforma Win32 e .NET com banco de dados Firebird e MSSQL.

Edinei Daniel Steffen

edineidaniel@gmail.com

É técnico e Bacharel em Sistemas de Informação, Analista e Desenvolvedor de Sistemas. Atua na área de Desenvolvimento de Softwares em Delphi para plataforma Win32 e .NET com banco de dados Firebird e MSSQL.

Conectividade, acessibilidade e agilidade são palavras que podem definir o novo modismo no que diz respeito ao desenvolvimento de softwares para controles industriais, tais como: de produção e PCP (*Programa de Controle de Produção*).

Nesse artigo iremos demonstrar de forma prática o uso da tecnologia *DataSnap* trabalhando com o protocolo *SOAP* (*Simple Object Access Protocol*). Ambas tecnologias estão disponíveis no *Delphi* e podem ser largamente utilizadas para os mais variados fins. O *DataSnap* é um conjunto de controles e conceitos que facilitam o desenvolvimento em camadas.

Resumo do DevMan

O desenvolvimento de sistemas em n camadas tem se popupalirazado no cenário de TI. Fazer com que uma aplicação possa ser gerenciada a distância via Web não é uma tarefa simples, porém o que veremos é a construção de uma solução completa utilizando o protocolo SOAP e o conceito DataSnap. Veremos que é perfeitamente possível colocar tudo em prática em apenas alguns passos.

A estrutura que será apresentada dará um enfoque exclusivo no uso da tecnologia SOAP e DataSnap para a integração de sistemas entre empresas, tratando do funcionamento e regras de negócio com algumas preocupações centralizadas de segurança.

Com essa facilidade, criar um sistema que possa ser acessado de qualquer parte do mundo acaba sendo uma das principais vantagens do *DataSnap*. Outra grande vantagem é, certamente, a velocidade tanto do desenvolvimento quanto do acesso aos dados. E velocidade conta bastante quando falamos em *Web* e sistemas integrados.

Sendo assim, criaremos uma aplicação que irá simular uma automação avançada entre indústrias do setor metal-mecânico e empresas fornecedoras de matérias-primas. A aplicação será desenvolvida em duas etapas, onde, nesse primeiro artigo, veremos desde o entendimento do problema até a criação e codificação da camada *Server*.

No segundo e último artigo da série, veremos a conexão do servidor por meio de duas aplicações cliente. Umas delas responsável pelos cadastros e gerenciamento do sistema e a outra que cuidará somente dos pedidos dos fornecedores. Faremos uma breve análise sobre o problema, passando ao levantamento de conceitos a serem utilizados no desenvolvimento, bem como proposição do banco de dados e o desenvolvimento da aplicação.

Entendimento do Problema

Para quem não sabe, uma linha de produção trabalha diariamente correndo alguns riscos, como por exemplo: falta de matéria-prima. Esses riscos certamente influenciam no faturamento da empresa. Quem controla, normalmente, os materiais e matérias-prima são gerentes de produção que tem que como principal função: organizar, cuidar e garantir a máxima eficiência de uma linha.

Um dos pontos-chave, para que uma linha de produção não sofra uma eventual parada por falta de algum tipo de material, gira em torno do controle e dos pedidos de compras. A principal regra para isso é justamente o controle do estoque mínimo de cada produto necessário. Tendo em vista essa situação, desenvolveremos um sistema capaz de detectar o limite mínimo de estoque e, automaticamente, emitir pedidos de compras avisando então os gerentes de produção.

Deste modo definimos que o sistema será desenvolvido em três camadas: *Camada do Banco de Dados*, *Camada do Servidor* (onde estarão as regras de negócio) e as *Camadas Cliente* que serão acessadas tanto para a empresa como para os fornecedores.

Client/Server x Três Camadas

Mas o que realmente muda em uma aplicação três ou *n* camadas para o método convencional (*Client/Server*)? Veja a análise dos dois casos.

- No método *duas camadas* (*Client/Server*): quando trabalhamos com eventos de componentes visuais, é fácil dispersarmos algumas regras como as consistências de campos. Isso pode acarretar em dores de cabeça no momento de uma manutenção, bem como alguns problemas ligados a distribuição do sistema. Se uma regra for mudada em um formulário, todo o sistema precisa ser *recompilado* e distribuído novamente.
- Analisando o método de três ou *n* camadas, podemos perceber que todas as regras de negócio localizam-se na camada de meio da aplicação, ou seja, a camada que fica entre o *banco de dados* e *cliente* junto ao servidor de aplicação.

Nota

DataSnap é uma coletânea de tecnologias que funcionam em conjunto e tem o objetivo de facilitar o desenvolvimento de aplicativos distribuídos.

Outra definição que também cabe citar aqui é de que *SOAP* foi definido em *XML*, para tanto a comunicação (remota) entre as camadas é baseada em *XML* e o transporte da mensagem é feita através do *HTTP* (*protocolo de transporte de dados na Internet*) permitindo que a comunicação aconteça sem bloqueios entre as camadas.

Banco de Dados

A **Figura 1** exibe de uma forma bem simples como se dará a organização e armazenamento de dados. Nele serão armazenadas as informações de produtos, fornecedores e pedidos com seus itens. Lembrando que os pedidos serão gerados automaticamente assim que os mesmos chegarem a seus limites mínimos de estoque para seus respectivos fornecedores.

O banco de dados definido para o desenvolvimento da aplicação foi o *Firebird 1.5* e conforme a **Figura 1** é proposto o *script* de criação do banco de dados demonstrado na **Listagem 1**. Utilize um gerenciador de banco de dados, como o *IBExpert* por exemplo, ou qualquer outro para executar o *script*.

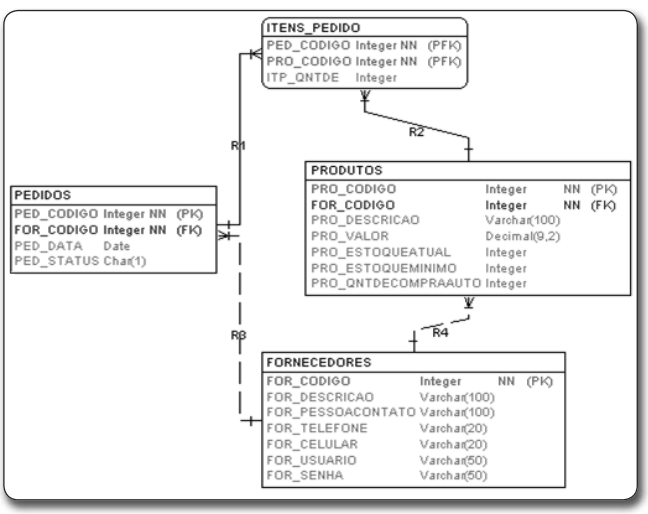

Figura 1. Estrutura proposta para o Banco de Dados

Listagem 1. Script para criação da base de dados

```
SET SQL DIALECT 3;
SET NAMES WIN1252;

SET CLIENTLIB '<caminho>\fbclient.dll';

CREATE DATABASE '<caminho>\BD_PEDIDOS.FDB'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 4096
DEFAULT CHARACTER SET WIN1252;

CREATE GENERATOR GEN_FORNECEDORES_ID;
SET GENERATOR GEN_FORNECEDORES_ID TO 0;

CREATE GENERATOR GEN_PEDIDOS_ID;
SET GENERATOR GEN_PEDIDOS_ID TO 0;

CREATE GENERATOR GEN_PRODUTOS_ID;
SET GENERATOR GEN_PRODUTOS_ID TO 0;

CREATE TABLE FORNECEDORES (
    FOR_CODIGO          INTEGER NOT NULL,
    FOR_DESCRICAO       VARCHAR(100),
    FOR_PESSOACONTATO   VARCHAR(100),
    FOR_TELEFONE        VARCHAR(20),
    FOR_CELULAR         VARCHAR(20),
    FOR_USUARIO         VARCHAR(50),
    FOR_SENHA           VARCHAR(50)
);

CREATE TABLE ITENS_PEDIDO (
    PED_CODIGO  INTEGER NOT NULL,
    PRO_CODIGO  INTEGER NOT NULL,
    ITP_QNTDE   INTEGER
);

CREATE TABLE PEDIDOS (
    PED_CODIGO  INTEGER NOT NULL,
    FOR_CODIGO  INTEGER NOT NULL,
    PED_DATA    DATE,
    PED_STATUS  CHAR(1)
);

CREATE TABLE PRODUTOS (
    PRO_CODIGO            INTEGER NOT NULL,
    FOR_CODIGO            INTEGER NOT NULL,
    PRO_DESCRICAO         VARCHAR(100),
    PRO_VALOR             NUMERIC(9,2),
    PRO_ESTOQUEATUAL      INTEGER,
    PRO_ESTOQUEMINIMO     INTEGER,
    PRO_QNTDECOMPRAAUTO   INTEGER
);

ALTER TABLE FORNECEDORES ADD PRIMARY KEY (FOR_CODIGO);
ALTER TABLE ITENS_PEDIDO ADD PRIMARY KEY (PED_CODIGO, PRO_CODIGO);
ALTER TABLE PEDIDOS ADD PRIMARY KEY (PED_CODIGO);
ALTER TABLE PRODUTOS ADD PRIMARY KEY (PRO_CODIGO);

SET TERM ^ ;

CREATE TRIGGER FORNECEDORES_BI FOR FORNECEDORES
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.FOR_CODIGO IS NULL) THEN
    NEW.FOR_CODIGO = GEN_ID(GEN_FORNECEDORES_ID,1);
END
^

CREATE TRIGGER PEDIDOS_BI FOR PEDIDOS
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.PED_CODIGO IS NULL) THEN
    NEW.PED_CODIGO = GEN_ID(GEN_PEDIDOS_ID,1);
END
^

CREATE TRIGGER PRODUTOS_BI FOR PRODUTOS
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.PRO_CODIGO IS NULL) THEN
    NEW.PRO_CODIGO = GEN_ID(GEN_PRODUTOS_ID,1);
END
^

SET TERM ; ^
```

Entendendo o funcionamento

O funcionamento da aplicação se dará baseado na **Figura 2**, simulando uma máquina cliente dentro da empresa que é responsável pela alimentação dos dados no sistema e realizar o controle de saída de materiais. Esta máquina pode estar na rede local da empresa ou em qualquer outro lugar na Internet. Já na máquina cliente do fornecedor, mediante um usuário e senha de identificação, apenas será disponibilizada uma interface para visualização dos pedidos que serão emitidos automaticamente pelo sistema central ou camada central onde se encontram as regras de negócio. Somente esta camada fará uma conexão com o banco de dados e disponibilizará as informações em um formato *XML* para as outras camadas.

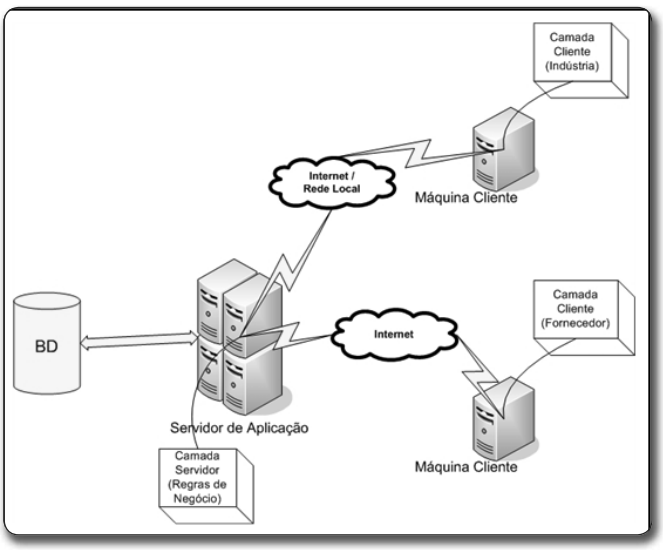

Figura 2. Visão gráfica do funcionamento da aplicação

Regras do Servidor

Como dito anteriormente, o nosso sistema será composto por algumas aplicações. Uma delas é nosso servidor de aplicação, que funcionará no servidor central da empresa. Para que possamos desenvolvê-lo, precisamos estabelecer algumas regras e então aplicá-las ao programa. Veja a seguir as principais regras de nosso sistema.

Nosso sistema deverá:

- Disponibilizar as informações para que o cliente possa realizar as manutenções de cadastros;
- Possibilitar abertura de pedidos para fornecedores;
- Incluir produtos em uma ordem de pedidos não fechada;
- Disponibilizar dados de pedidos por fornecedor;
- Finalizar pedidos emitidos para um fornecedor;
- Atualizar estoque dos produtos;
- Identificar fornecedor pelo usuário e senha do banco de dados.

Perceba que nossas regras na verdade são tarefas que deverão ser realizadas pelo servidor de aplicação. Em outras palavras, ele servirá as aplicações cliente como um *executor de tarefas*. É ele quem receberá as ordens vindas de cada aplicação e executará no servidor. Essa é a essência básica de um servidor de aplicação.

Desenvolvimento da solução proposta

Após a montagem do cenário no qual será desenvolvida a solução, iniciaremos com a criação da camada de *negócios*, onde serão desenvolvidos todos os procedimentos e regras para o funcionamento das outras camadas. Para o desenvolvimento será utilizado o *CodeGear Rad Studio 2007 for Win32*, mas qualquer outra versão do Delphi poderá ser utilizada.

No *RAD 2007*, em *File>New>Other>Delphi Projects> WebServices>SOAP Server Application*. Após escolhida esta opção deverá ser definido o tipo de estrutura *Web* para o sistema, no caso, neste exemplo iremos utilizar *ISAPI/NSAPI Dynamic Link Library*, conforme ilustrado na **Figura 3**.

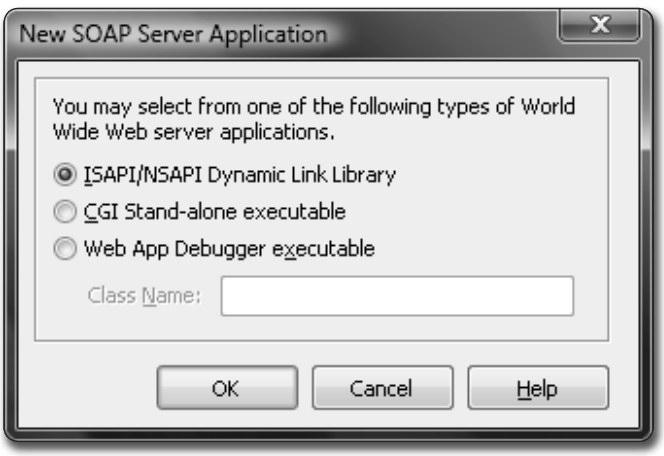

Figura 3. Criando a aplicação WebServices

Definido o tipo da aplicação, surgirá uma caixa de diálogo com a pergunta *Create Interface for SOAP module?*. Responda que não (*No*) para que não seja criada uma interface para métodos públicos do serviço *SOAP*. Faremos isso em outro momento.

Salve os arquivos da aplicação (*File>Save All*) deixando a *unit* principal como *uWMPedidos.pas* e o projeto como *PedidosSoapServer.dpr*. Após salvar o projeto, perceba que a Unit1, salva como uWMPedidos.pas, é na verdade um *WebModule*. Modifique sua propriedade *Name* para *WMPedidos* e deixe o restante da estrutura como foi criada pelo Delphi. Em seguida acesse menu *File>New>Other>Delphi Projects>WebServices>SOAP Server Data Module*. Na janela que aparece, dê o nome de *DMPedidos* ao *DataModule* e em seguida salve-o como *uDMPedidos.pas*.

A estrutura inicial da camada servidora está configurada, agora é necessário adicionar os componentes que farão a interação com o banco de dados. Todos os componentes que utilizaremos serão adicionados no *DMPedidos*.

Iniciando com o componente *SQLConnection* (*sqlConexao*) da paleta *dbExpress* para viabilizar a conexão com o banco de dados. Acesse sua propriedade *Params* configurando-a de acordo com informações da **Figura 4**. Aproveite e configure a propriedade *Login Prompt* para *False*.

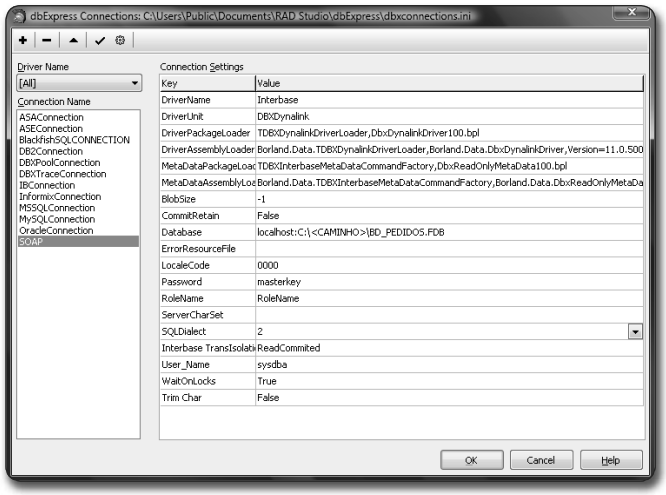

Figura 4. Configuração do componente de conexão com o banco de dados

Realizada a configuração do componente de conexão com o banco, passaremos a utilizar os componentes para acessar e manipular os dados das diversas tabelas do banco de dados.

Os componentes indicados para este fim serão o *SQLQuery e DataSetProvider* das paletas *dbExpress* e *DataAccess*, respectivamente.

De imediato adicione cinco componentes *SQLQuery* (*sqlProdutos*, *sqlFornecedores*, *sqlPedidos*, *sqlItens_Pedido* e *sqlPedidosPorFornecedor*) e na propriedade *SQLConnection* de cada componente, selecione o componente de conexão e em suas propriedades *SQL* adicione os comandos *SQL`s*, na ordem em que aparecem na **Listagem 2.**

Note que alguns comandos *SQL* possuem um parâmetro para ser informado antes da abertura ou execução do comando no banco de dados. É muito importante fazer o uso de parâmetros para resgatar os dados de uma tabela principalmente quando se trata de uma estrutura de comunicação *HTTP*, fazendo com que não haja a necessidade de toda a vez que for solicitado uma consulta se retornar todos os dados da tabela. A presença da cláusula *WHERE* na instrução *SQL* se torna indispensável para este tipo aplicação.

Devido à presença de um parâmetro na *SQL*, devemos também configurar o tipo de dado que deve ser informado nesse parâmetro. Clicando na propriedade *Params* e selecionando o parâmetro desejado, altere a propriedade *DataType* para *ftString* para valores do tipo texto ou *ftInteger* para tipos de dados inteiros.

Ative os componentes marcando como *True* a propriedade *Active* de cada componente. Abra o *FieldsEditor* de cada *Query* e adicione todos os campos clicando com botão direito do mouse e escolhendo a opçãp *Add all Fields*.

Algo que também deve ser observado nesta configuração são os *ProviderFlags*. Essa propriedade permite que possamos configurar quais os campos fazem parte ou não da chave primária da tabela além, é claro, de informar ao *DataSetProvider* o que fazer com cada campo enviado ou recebido pela *Query*. Os campos que fazem parte da chave primária da tabela devem ter a seguinte configuração na propriedade *ProviderFlags*:

Listagem 2. Comandos SQL

```
[sqlProdutos]
SELECT
  PRODUTOS.PRO_CODIGO, PRODUTOS.FOR_CODIGO,
  PRODUTOS.PRO_DESCRICAO, PRODUTOS.PRO_VALOR, PRODUTOS.PRO_ESTOQUEATUAL,
  PRODUTOS.PRO_ESTOQUEMINIMO, PRODUTOS.PRO_QNTDECOMPRAAUTO
FROM
  PRODUTOS
WHERE
  (PRODUTOS.PRO_DESCRICAO LIKE :PDESCRICAO)
ORDER BY
  PRODUTOS.PRO_DESCRICAO

[sqlFornecedores]
SELECT
  FORNECEDORES.FOR_CODIGO, FORNECEDORES.FOR_DESCRICAO,
  FORNECEDORES.FOR_PESSOACONTATO, FORNECEDORES.FOR_TELEFONE,
  FORNECEDORES.FOR_CELULAR, FORNECEDORES.FOR_USUARIO,
  FORNECEDORES.FOR_SENHA
FROM
  FORNECEDORES
WHERE
  (FORNECEDORES.FOR_DESCRICAO LIKE :PFOR_DESCRICAO)
ORDER BY
  FORNECEDORES.FOR_DESCRICAO

[sqlPedidos]
SELECT
  PEDIDOS.PED_CODIGO, PEDIDOS.FOR_CODIGO, PEDIDOS.PED_DATA,
  PEDIDOS.PED_STATUS
FROM
  PEDIDOS
ORDER BY
  PEDIDOS.PED_DATA

[sqlItens_Pedido]
SELECT
  ITENS_PEDIDO.PED_CODIGO, ITENS_PEDIDO.PRO_CODIGO,
  ITENS_PEDIDO.ITP_QNTDE, PRODUTOS.PRO_DESCRICAO
FROM
  ITENS_PEDIDO
  INNER JOIN PRODUTOS ON
  (ITENS_PEDIDO.PRO_CODIGO = PRODUTOS.PRO_CODIGO)
WHERE
  (ITENS_PEDIDO.PED_CODIGO = :PPED_CODIGO)
ORDER BY
  PRODUTOS.PRO_DESCRICAO

[sqlPedidosPorFornecedor]
SELECT
  PEDIDOS.PED_CODIGO, PEDIDOS.FOR_CODIGO, PEDIDOS.PED_DATA,
  PEDIDOS.PED_STATUS
FROM
  PEDIDOS
WHERE
  (PEDIDOS.FOR_CODIGO = :PFOR_CODIGO) AND (PEDIDOS.PED_STATUS = 'A')
ORDER BY
  PEDIDOS.PED_DATA
```

[pfInUpdate, prInWhere, pfInKey]. E para os demais campos deve somente estar ativada a *flag*: *[pfInUpdate]*.

Necessitamos também de cinco componentes *DataSetProvider (dspProdutos, dspFornecedores, dspPedidos, dspItens_Pedido, dspPedidosPorFornecedor)* da paleta *DataAccess* relacionados, através da propriedade *DataSet*, com os respectivos componentes *SQLQuery* (*sqlProdutos, sqlFornecedores, sqlPedidos, sqlItens_Pedido, sqlPedidosPorFornecedor*) e na propriedade *UpdateMode* selecione *upWhereKeyOnly*.

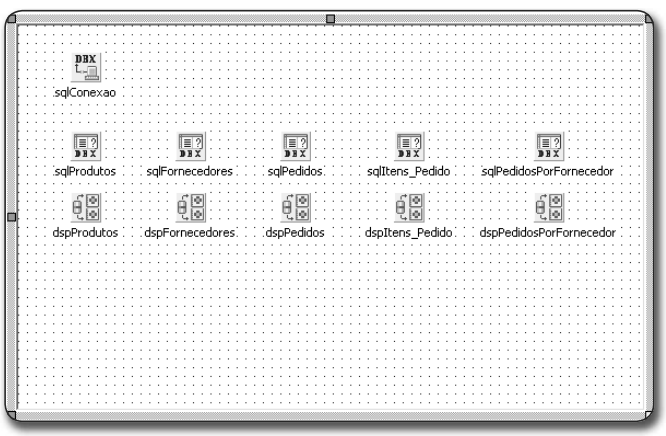

Figura 5. Relação de componentes necessários

Configurados os componentes de acesso e manipulação dos dados do sistema no servidor da nossa aplicação, precisamos agora criar a interface que fará a comunicação com as aplicações cliente e que farão acesso através do protocolo *SOAP*. Na interface do servidor serão criadas as funções e métodos que cuidarão das regras de negócio, ou seja, procedimentos de cadastros, pesquisa, verificações, validações, segurança, entre outros.

No menu *File>New>Other>Delphi Projects>WebServices>SOAP Server Interface* (**Figura 6**), que ao ser executado irá solicitar que o nome do serviço de interface seja informado. Chame o serviço de *funcoes* e em seguida desmarque o item *Generate comments* e confirme o diálogo criando assim dois novos arquivos de código no servidor (*funcoesImpl.pas* e *funcoesIntf.pas*). Sendo eles a interface visível através de conexões realizadas e outro com as implementações de código que ficarão centralizadas nesta camada *Server*.

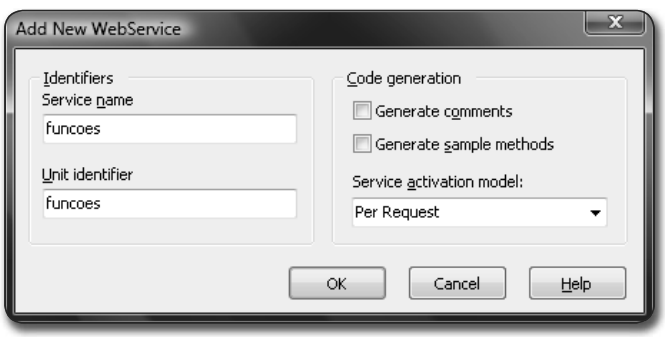

Figura 6. Criando a interface da camada *server*

A *unit funcoesIntf.pas* será responsável por exportar, ou seja, disponibilizar e tornar visível à *Web* as funções a serem invocadas pela camada servidora. Nessa *unit* faremos somente as declarações de cada função, pois a implementação de cada uma será feita em outra *unit*. Ao final de cada *function/procedure* declarada iremos adicionar *stdcall* indicando que os parâmetros serão exportados.

Declare as funções e procedimentos de acordo com o código da **Listagem 3**, onde encontram-se listados todas as *functions* e *procedures* necessárias para a manipulação da camada servidora através da conexão de camadas clientes.

Listagem 3. Implementando os códigos do server

```
unit funcoesIntf;

interface

uses
  InvokeRegistry, Types, XSBuiltIns;

type
  Ifuncoes = interface(IInvokable)
  ['{4389A2ED-61D6-4A59-AC50-2F9DFDC7CDC9}']
  function MontaPedidos(): String; stdcall;
  function AtualizarEstoqueProduto(CodigoProduto, Qntde, Operacao:
    String): Boolean; stdcall;
  function AlterarStatusPedido(CodigoPedido, Status: String):
    string; stdcall;
  function IdentificaFornecedor(Usuario, Senha: String;
    var Codigo, Nome: String): boolean; stdcall;
  end;

implementation

initialization
  InvRegistry.RegisterInterface(TypeInfo(Ifuncoes));
end.
```

Observe que estamos realizando a exportação de quatro funções responsáveis pela montagem da lista de novos pedidos, atualização de quantidades de produtos solicitadas no estoque, alteração de *status* de pedidos já existentes e também a identificação do fornecedor solicitante através de usuário e senha fornecidos. Os parâmetros para estas funções serão trafegados de forma criptografada, dificultando a intervenção de invasores na rede e captura ou alteração das informações.

Partindo agora para a implementação das funções na *unit funcoesImpl.pas*. Temos a classe *TInvokableClass* que é a responsável pela estrutura de todas as classes que implementam interfaces *IInvokable*. Esta classe é semelhante à classe *TInterfacedObject* usada em aplicações *COM*. Nesta *unit* é que faremos todas as regras da nossa camada *Server*, ou seja, vamos implementar as funções a serem exportadas e além desta, criar outras para que façam um gerenciamento silencioso de todos os processos que são executados.

Faça a declaração das funções e procedimentos conforme a **Listagem 4**. Perceba que na linha 8 fiz a declaração de uma constante chamada *SegKey*. Essa constante será nossa chave de segurança, como um *Token*. Isso evita que alguém acesse o banco de dados diretamente por outros meios. Para que o Delphi crie a estrutura de cada *function/procedure* pressione simultaneamente *Ctrl + Shift + C*.

Antes de detalharmos cada função, vejamos resumidamente a função que cada uma desempenhará no sistema. Na sessão *private* temos declaradas as funções/procedimentos: *MontaPedidoFornecedor, AdicionaProdutoPedido, VerificaPedidoProduto, VerificaPedidoAbertoFornecedor* e *Seguranca*. Veja a seguir a descrição resumida delas:

- *MontaPedidoFornecedor*: procedimento auxiliar utilizado na montagem de pedidos para fornecedores, fazendo que com seja possível abrir pedidos por fornecedor de acordo com o valor que é atribuído ao parâmetro;
- *AdicionaProdutoPedido*: procedimento responsável por adicionar um produto ao pedido criado ou em edição (aberto), sendo necessário informar o código do mesmo, o código do produto e a quantidade a ser solicitada;
- *VerificaPedidoProduto*: função responsável por verificar se um determinado produto já se encontra em algum pedido que está com seu *status Aberto*, retornando um valor *booleano (Verdadeiro/Falso)*;
- *VerificaPedidoAbertoFornecedor*: como o próprio nome já sugere, esta função é responsável pela verificação de um pedido que está *Aberto* para uma determinado fornecedor. Retorna ocorrências do tipo *True* ou *False* de acordo com o resultado da busca.
- *Seguranca*: função responsável pela criptografia e descriptografia das informações que serão trafegadas entre as aplicações clientes e servidor. A função deverá receber como parâmetros a ação *E=Criptografar, D=Descriptografar*, o texto a ser transformado e a chave que é utilizada na criptografia dos caracteres. Esse último parâmetro é justamente nossa constante *SegKey*.

Listagem 4. Declaração das funções do servidor

```
01 unit funcoesImpl;
02
03 interface
04
05 uses
06   InvokeRegistry, Types, XSBuiltIns, funcoesIntf, SQLExpr;
07
08 const SegKey = 'SOAP';
09
10 type
11   { Tfuncoes }
12   Tfuncoes = class(TInvokableClass, Ifuncoes)
13   private
14     procedure MontaPedidoFornecedor(Fornecedor: String);
15     procedure AdicionaProdutoPedido(Pedido, Produto, Qntde:
16       string);
17     function VerificaPedidoProduto(Fornecedor, Produto:
         String): boolean;
18     function VerificaPedidoAbertoFornecedor(Fornecedor:
         String): String;
19     function Seguranca(Acao, Texto, Chave: WideString):
         WideString;
21   public
22     function MontaPedidos(): string; stdcall;
23     function AtualizarEstoqueProduto(CodigoProduto, Qntde,
         Operacao:
24       string): Boolean; stdcall;
25     function AlterarStatusPedido(CodigoPedido, Status:
         string): string; stdcall;
26     function IdentificaFornecedor(Usuario, Senha: string; var
         Codigo, Nome:
27       string): boolean; stdcall;
28   end;
```

A sessão *public* possui as mesmas funções declaradas na *unit funcoesIntf.pas*. Isso acontece porque nossa classe atual é herdada de *Ifuncoes*, veja:

```
Tfuncoes = class(TInvokableClass, Ifuncoes)
```

A maior vantagem em se trabalhar com interfaces, é que elas funcionam como contratos nos obrigando a ter exatamente a mesma estrutura que a interface mãe. Em outras palavras, todas as funções e procedimentos declarados em *Ifuncoes* devem ser implementadas *Tfuncoes*, nossa classe atual. Isso facilita o trabalho e evita que cada desenvolvedor na equipe, crie suas próprias funções sem respeitar a hierarquia de ações.

Nós precisamos declarar então *MontaPedidos, AtualizaEstoqueProduto, AlterarStatusPedido* e *IdentificaFornecedor*. Veja a descrição resumida de cada função/procedimento:

- *MontaPedidos*: função responsável pela montagem dos pedidos com os produtos cujo seus estoques atuais estão iguais ou abaixo do estoque mínimo configurado no momento do cadastro do produto. Montando uma lista de produtos por fornecedor, o sistema realiza ainda verificações se já existe um pedido aberto para o fornecedor adicionando mais um produto à lista, caso contrário, um novo pedido é criado;
- *AtualizaEstoqueProduto*: função responsável pela atualização da quantidade em estoque do produto. É utilizada no momento de aumentar ou diminuir a quantidade, dependendo do valor informado na operação (*soma, subtração*), tendo como retorno um valor *verdadeiro* ou *falso* para identificar o sucesso ou não da operação;
- *AlterarStatusPedido*: função utilizada para a atualização do *status* do pedido. Cada fornecedor terá seu sistema *cliente* com a lista de pedidos que estejam abertos. Assim, a cada pedido que for encaminhado ao servidor, o *Status* desse pedido deve ser alterado para não correr o risco de um produto estar na lista e não ser enviado;
- *IdentificaFornecedor*: função responsável pela validação do *login* dos fornecedores junto a seus sistemas, retornando seu respectivo código e nome por parâmetro de retorno;

Codificando e entendendo o sistema

Aparentemente o nosso sistema parece ser bastante complexo, mas veremos nesse tópico que na verdade todo o processo de desenvolvimento com *SOAP* é bem simples e objetivo. No tópico anterior pressionamos Ctrl + Shift + C para que o Delphi nos criasse os cabeçalhos de cada função/procedimento. Vamos agora codificar cada um e entender seu funcionamento. Comecemos pelo procedimento *AdicionaProdutoPedido* (**Listagem 5**).

No início do procedimento (*Linhas 3* e *4*) declaramos duas variáveis que serão responsáveis por receberem os ponteiros de memória para o componente *SQLDataSet* e o *DataModule DMPedidos* que criaremos em *runtime*. Perceba que os criamos nas linhas 7 e 8. Em seguida atualizamos as propriedades do nosso *SQLDataSet* (*SQL*) utilizando um bloco *with..do* (*Linha 10*) para facilitar nosso trabalho de digitação dos procedimentos. Note que atualizamos a propriedade *SQLConnection* apontando-a para o *DataModule DMPedidos* e então modificamos sua propriedade *CommandText* incluindo uma instrução *SQL* que fará a inclusão (*INSERT*) dos itens de pedido na tabela de mesmo nome (*Linhas 14* e *15*). Na seqüência atualizamos os parâmetros *PED_CODIGO, PRO_CODIGO* e *ITP_QNTDE* passando o conteúdo das variáveis *Pedido, Produto* e *Qntde* recebidas por parâmetro pelo procedimento (*Linhas 16* a *18*). Por fim, chamamos o método *ExecSQL* (*Linha 19*) que faz a execução do comando *SQL* no servidor.

Todo esse procedimento é envolvido em um bloco *try..finally* (*Linhas 9, 21* e *25*). Isso evita que um erro ocorra e os objetos *SQL* e *DM* não sejam liberados da memória.

Nota do DevMan

É extremamente importante fazer o controle de transação em operações dessa natureza em toda e qualquer aplicação. Para fazer o controle de transação utilizando DBExpress de maneira mais simples, basta criarmos uma variável do tipo TransactionDesc e atribuí-la aos métodos StartTransaction, Commit e RollBack do SQLConnection. Além disso precisamos atribuir um ID a variável e informar qual o nível de isolamento da transação. Veja a seguir um exemplo:

```
procedure TfrmProcesso.btnGravarClick(Sender: TObject);
var
 Transacao: TTransactionDesc;
begin
 try
  Transacao.TransactionID := 1;
  Transacao.IsolationLevel := xilREADCOMMITTED;
  SQLConnection.StartTransaction(Transacao);
  { Procedimentos de gravação }
  SQLConnection.Commit(Transacao);
 except
  SQLConnection.Rollback();
 end;
end;
```

Perceba que isolamos o procedimento em um bloco try..except o que significa que caso haja um problema, toda a operação será cancelada pelo método RollBack.

Listagem 5. Procedimento AdicionaProdutoPedido

```
01 procedure Tfuncoes.AdicionaProdutoPedido(Pedido, Produto,
    Qntde: String);
02 var
03   sql: TSQLDataSet;
04   DM : TDMPedidos;
05 begin
06   try
07     sql := TSQLDataSet.Create(nil);
08     DM := TDMPedidos.Create(nil);
09     try
10       with sql do
11       begin
12         SQLConnection := DM.sqlConexao;
13         CommandText :=
14           'INSERT INTO ITENS_PEDIDO (PED_CODIGO,PRO_
               CODIGO,ITP_QNTDE) '+
15           ' VALUES (:PED_CODIGO,:PRO_CODIGO,:ITP_QNTDE) ';
16         ParamByName('PED_CODIGO').AsString := Pedido;
17         ParamByName('PRO_CODIGO').AsString := Produto;
18         ParamByName('ITP_QNTDE').AsString  := Qntde;
19         ExecSQL();
20       end;
21     finally
23       sql.free;
24       DM.free;
25     end;
26  except
27  end;
28 end;
```

Como vimos, não há segredos no procedimento *AdicionaProdutoPedido*. Agora, vejamos outro procedimento importante de nosso sistema. A *procedure AlterarStatusPedido* tem como principal função alterar o estado do pedido como seu próprio nome sugere. Logo de início podemos ver que o procedimento começa bem semelhante ao procedimento

anterior, ou seja, declarando as variáveis que serão usadas para instanciar um *SQLDataSet* e o *DMPedidos*. Saltamos então para as *linhas 14* até *21*. Nelas montamos uma instrução *SQL* para atualizar (UPDATE) o pedido em questão. Atualizamos os parâmetros *PPED_STATUS* e *PPED_CODIGO* usando o valor que veio nos parâmetros da *function* e por fim chamamos o método *ExecSQL* para executar o comando. A novidade aqui é que criamos uma *function* para esse processo, assim podemos receber o estado final da operação, ou seja, se foi ou não bem sucedido (*Linha 21*) (**Listagem 6**).

Listagem 6. Procedimento AlterarStatusPedido

```
01 function Tfuncoes.AlterarStatusPedido(CodigoPedido, Status:
    String): string;
02 var
03   sql: TSQLDataSet;
04   DM : TDMPedidos;
05 begin
06   result := '';
07   try
08     sql := TSQLDataSet.Create(nil);
09     DM := TDMPedidos.Create(nil);
10     try
11       with sql do
12       begin
13         SQLConnection := DM.sqlConexao;
14         CommandText :=
15           ' UPDATE PEDIDOS '+
16           ' SET PED_STATUS = :PPED_STATUS  '+
17           ' WHERE (PED_CODIGO = :PPED_CODIGO) ';
18         ParamByName('PPED_STATUS').AsString :=
            Seguranca('D',Status,SegKey);
19         ParamByName('PPED_CODIGO').AsString := Seguranca('D',
            CodigoPedido,SegKey);
20         ExecSQL();
21         Result := 'Pedido atualizado com sucesso!';
22       end;
23     finally
24       sql.free;
25       DM.free;
26     end;
27   except
28     Result := 'Problemas com a atualização do Pedido!';
29   end;
30 end;
```

Outra função importante de nosso sistema tem o nome de *AtualizarEstoqueProduto*. Após praticamente todas as operações no banco de dados, o estoque deve ser consultado e atualizado para evitar surpresas no futuro. Essa função tem exatamente essa responsabilidade. Veja sua implementação na **Listagem 7**

Vamos novamente pular diretamente para as linhas 15 a 23 já que o início de nossa função já é conhecido. Aqui estamos montando uma instrução SQL para atualizar o banco de dados com a quantidade de estoque restante. Repare que informamos na cláusula *Where* de nossa instrução o código do produto (*PRO_CODIGO*) e o estoque atual (*PRO_ESTOQUEATUAL*). O último parâmetro como sempre é a chave de validação, o *SegKey*. Então executamos o procedimento e resultamos em *True* ou *False*.

A *procedure MontaPedidoFornecedor* tem o mesmo princípio das demais funções que já vimos. Sua responsabilidade principal é incluir o pedido de compras no banco de dados, por isso veja sua instrução SQL nas linhas *14* e *15* da **Listagem 8**. O resumo é que apenas informamos o código do fornecedor e o pedido é incluído no sistema. O segundo e o terceiro parâmetro são a data e o estado do pedido, respectivamente.

Listagem 7. Código da função de atualização do estoque dos produtos

```
01 function Tfuncoes.AtualizarEstoqueProduto(CodigoProduto, Qntde,
02   Operacao: String): Boolean;
03 var
04   sql: TSQLDataSet;
05   DM : TDMPedidos;
06 begin
07   result := false;
08   try
09     sql := TSQLDataSet.Create(nil);
10     DM := TDMPedidos.Create(nil);
11     try
12       with sql do
13       begin
14         SQLConnection := DM.sqlConexao;
15         CommandText :=
16           'UPDATE PRODUTOS '+
17           ' SET PRO_ESTOQUEATUAL = PRO_ESTOQUEATUAL ' + Operacao +
18           ' :PRO_ESTOQUEATUAL  '+
19           ' WHERE (PRO_CODIGO = :PRO_CODIGO) ';
20         ParamByName('PRO_ESTOQUEATUAL').AsString :=
            Seguranca('D',Qntde,SegKey);
21         ParamByName('PRO_CODIGO').AsString :=
22           Seguranca('D',CodigoProduto,SegKey);
23         ExecSQL();
24         result := true;
25       end;
26     finally
27       sql.free;
28       DM.free;
29     end;
30   except
31     result := false;
32   end;
33 end;
```

Listagem 8. Procedimento de inclusão de pedidos

```
01 procedure Tfuncoes.MontaPedidoFornecedor(Fornecedor: String);
02 var
03   sql: TSQLDataSet;
04   DM : TDMPedidos;
05 begin
06   try
07     sql := TSQLDataSet.Create(nil);
08     DM := TDMPedidos.Create(nil);
09     try
10       with sql do
11       begin
12         SQLConnection := DM.sqlConexao;
13         CommandText :=
14           'INSERT INTO PEDIDOS (FOR_CODIGO,PED_DATA,PED_STATUS) '+
15           'VALUES (:FOR_CODIGO,''TODAY'',''A'')';
16         Params[0].AsString := Fornecedor;
17         ExecSQL();
18       end;
19     finally
20       sql.free;
21       DM.free;
22     end;
23   except
24   end;
25 end;
```

A montagem do pedido propriamente dito é um pouco mais extensa, mas não é complexa. Veja na **Listagem 9** toda sua implementação. Nas *linhas 17* e *18* está a instrução *SQL* responsável por resgatar os dados produtos. Logo em seguida verificamos se a consulta retornou dados (*IsEmpty*) o que significa que existe o produto consultado. Na seqüência envolvemos a operação em um laço *while..do* para que possamos incluir cada produto ao pedido. Perceba que chamamos as funções *VerificaPedidoAbertoFornecedor* e *MontaPedidoFornecedor*. As funções fazem a checagem e montagem dos pedidos do fornecedor, respectivamente. Por serem funções extensas, não faremos a implementação delas aqui. Elas poderão ser vistas baixando o código-fonte do site da *DevMedia* na sessão *ClubeDelphi*. Por fim chamamos *VerificaPedidoProduto* e *AdicionaPedidoProduto*, que já foram explicadas anteriormente.

Listagem 9. Montagem dos Pedidos

```
01 function Tfuncoes.MontaPedidos(): string;
02 var
03   sql: TSQLDataSet;
04   DM : TDMPedidos;
05   CodigoPedido: String;
06 begin
07   result := '';
08   CodigoPedido := '0';
09   try
10     sql := TSQLDataSet.Create(nil);
11     DM := TDMPedidos.Create(nil);
12     try
13       with sql do
14       begin
15         SQLConnection := DM.sqlConexao;
16         CommandText :=
17           'SELECT PRO_CODIGO, FOR_CODIGO, PRO_QNTDECOMPRAAUTO
               FROM PRODUTOS '+
18           'WHERE PRO_ESTOQUEATUAL <= PRO_ESTOQUEMINIMO ';
19         Open;
20         if not IsEmpty then
21           while not EOF do
22           begin
23             CodigoPedido :=
24               VerificaPedidoAbertoFornecedor(
25                 FieldByName('FOR_CODIGO').AsString);
26             if (CodigoPedido = '0') then
27             begin
28               MontaPedidoFornecedor(FieldByName('FOR_
                   CODIGO').AsString);
29               CodigoPedido :=
30                 VerificaPedidoAbertoFornecedor(
31                   FieldByName('FOR_CODIGO').AsString);
32             end;
33             if not VerificaPedidoProduto(FieldByName('FOR_
                CODIGO').AsString,
34               FieldByName('PRO_CODIGO').AsString) then
35             AdicionaProdutoPedido(CodigoPedido,
36               FieldByName('PRO_CODIGO').AsString,
37               FieldByName('PRO_QNTDECOMPRAAUTO').AsString);
38             Next;
39           end;
40         Result := 'Pedidos encaminhados aos fornecedores com
              sucesso!';
41       end;
42     finally
43       sql.free;
44       DM.free;
45     end;
46   except
47     Result := 'Problemas na montagem dos pedidos!';
48   end;
49 end;
```

Todas nossas regras da camada servidora estão na *unit* recém codificada, onde as funções e procedimentos presentes serão responsáveis pelo gerenciamento das informações, ou seja, todos os métodos estarão presentes nesta camada, caracterizando assim nossa camada de negócios. Todos os procedimentos e funções foram escritos manualmente, sem utilização de componentes, exceto a ligação com a base de dados onde herdamos o *SQLConnection* configurado anteriormente no *SOAP Server Data Module (DMPedidos)*.

Basicamente, todas as *functions* e *procedures* seguem o mesmo padrão de programação, como a criação de um componente do tipo *TSQLDataSet* em *runtime*, responsável pela execução dos comandos *SQL* retornando resultados de buscas ou ações realizadas. Após a execução das instruções, um *result* é atribuído às *functions*, e em seguida, os objetos criados em memória são liberados utilizando o *free*.

A partir destas funções, terminamos a idéia de desenvolvimento da aplicação *Server*. Ao compilarmos a o projeto será gerada uma DLL que poderá ser disponibilizada e hospedada em um servidor Web para que possa ser acessada através do protocolo HTTP. Hospedado o arquivo já poderemos iniciar o processo de codificação das aplicações da camada cliente.

Conclusão

Certamente o desenvolvimento em camadas tem crescido nos últimos tempos por ser mais escalável e de fácil implementação e manutenção. Como vimos, não há segredos ou complexidade no desenvolvimento da camada servidora. Criamos o banco de dados, desenvolvemos as funções que serão acessada pelas camadas cliente e geramos a DLL para distribuição.

Na próxima e última fase desse mini-curso, veremos como montar a parte cliente e fazer a distribuição da aplicação. Baseado nesta estrutura, você poderá aplicar outros conceitos úteis no gerenciamento de pedidos. Poderá ainda trabalhar na segurança necessária entre as camadas através do protocolo SOAP.

No próximo artigo faremos a conclusão do artigo criando os dois aplicativos que utilizaremos na camada cliente, deixando o sistema funcional. Abraço e até a próxima. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback